Chapter 5 Adversarial Search and Games In which we explore environments where other agents are plotting against us. In this chapter we cover competitive environments, in which two or more agents have conflicting goals, giving rise to adversarial search problems. Rather than deal with the chaos of real-world skirmishes, we will concentrate on games, such as chess, Go, and poker. For AI researchers, the simplified nature of these games is a plus: the state of a game is easy to represent, and agents are usually restricted to a small number of actions whose effects are defined by precise rules. Physical games, such as croquet and ice hockey, have more complicated descriptions, a larger range of possible actions, and rather imprecise rules defining the legality of actions. With the exception of robot soccer, these physical games have not attracted much interest in the AI community. Adversarial search5.1 Game Theory There are at least three stances we can take towards multi-agent environments. The first stance, appropriate when there are a very large number of agents, is to consider them in the aggregate as an economy, allowing us to do things like predict that increasing demand will cause prices to rise, without having to predict the action of any individual agent. Economy Second, we could consider adversarial agents as just a part of the environment—a part that makes the environment nondeterministic. But if we model the adversaries in the same way that, say, rain sometimes falls and sometimes doesn't, we miss the idea that our adversaries are actively trying to defeat us, whereas the rain supposedly has no such intention. The third stance is to explicitly model the adversarial agents with the techniques of adversarial game-tree search. That is what this chapter covers. We begin with a restricted class of games and define the optimal move and an algorithm for finding it: minimax search, a generalization of AND–OR search (from Figure 4.11). We show that pruning makes the search more efficient by ignoring portions of the search tree that make no difference to the optimal move. For nontrivial games, we will usually not have enough time to be sure of finding the optimal move (even with pruning); we will have to cut off the search at some point. Pruning For each state where we choose to stop searching, we ask who is winning. To answer this question we have a choice: we can apply a heuristic evaluation function to estimate who is winning based on features of the state (Section 5.3), or we can average the outcomes of many fast simulations of the game from that state all the way to the end (Section 5.4). Section 5.5 discusses games that include an element of chance (through rolling dice or shuffling cards) and Section 5.6 covers games of imperfect information (such as poker and bridge, where not all cards are visible to all players). Imperfect information 5.1.1 Two-player zero-sum games The games most commonly studied within AI (such as chess and Go) are what game theorists call deterministic, two-player, turn-taking, perfect information, zero-sum games. "Perfect information" is a synonym for "fully observable," and "zero-sum" means that what is good for one player is just as bad for the other: there is no "win-win" outcome. For games we often use the term move as a synonym for "action" and position as a synonym for "state." 1 Some authors make a distinction, using "imperfect information game" for one like poker where the players get private information about their own hands that the other players do not have, and "partially observable game" to mean one like StarCraft II where each player can see the nearby environment, but not the environment far away. Perfect information Zero-sum games Position   1Move We will call our two players MAX and MIN, for reasons that will soon become obvious. MAX moves first, and then the players take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined with the following elements: : The initial state, which specifies how the game is set up at the start. TO-MOVE : The player whose turn it is to move in state . ACTIONS : The set of legal moves in state . RESULT : The transition model, which defines the state resulting from taking action in state . Transition model IS-TERMINAL : A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states. Terminal test Terminal state UTILITY : A utility function (also called an objective function or payoff function), which defines the final numeric value to player when the game ends in terminal state . In chess, the outcome is a win, loss, or draw, with values , 0, or . Some games S0 (s) s (s) s (s, a) a s (s) (s, p) p s 1 1/2 2 ⍰have a wider range of possible outcomes—for example, the payoffs in backgammon range from 0 to . 2 Chess is considered a "zero-sum" game, even though the sum of the outcomes for the two players is for each game, not zero. "Constant sum" would have been a more accurate term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of . Much as in Chapter 3 , the initial state, ACTIONS function, and RESULT function define the state space graph—a graph where the vertices are states, the edges are moves and a state might be reached by multiple paths. As in Chapter 3 , we can superimpose a search tree over part of that graph to determine what move to make. We define the complete game tree as a search tree that follows every sequence of moves all the way to a terminal state. The game tree may be infinite if the state space itself is unbounded or if the rules of the game allow for infinitely repeating positions. State space graph Search tree Game tree Figure 5.1 shows part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and 192 +1 1/2 ⍰ ⍰ ⍰MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three squares in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are good for MAX and bad for MIN (which is how the players get their names). Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game. For tic-tac-toe the game tree is relatively small—fewer than terminal nodes (with only 5,478 distinct states). But for chess there are over nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world. 9! = 362,880 10 405.2 Optimal Decisions in Games MAX wants to find a sequence of actions leading to a win, but MIN has something to say about it. This means that MAX's strategy must be a conditional plan—a contingent strategy specifying a response to each of MIN's possible moves. In games that have a binary outcome (win or lose), we could use AND–OR search (page 125) to generate the conditional plan. In fact, for such games, the definition of a winning strategy for the game is identical to the definition of a solution for a nondeterministic planning problem: in both cases the desirable outcome must be guaranteed no matter what the "other side" does. For games with multiple outcome scores, we need a slightly more general algorithm called minimax search. Minimax search Consider the trivial game in Figure 5.2 . The possible moves for MAX at the root node are labeled , , and . The possible replies to for MIN are , , , and so on. This particular game ends after one move each by MAX and MIN. (NOTE: In some games, the word "move" means that both players have taken an action; therefore the word ply is used to unambiguously mean one move by one player, bringing us one level deeper in the game tree.) The utilities of the terminal states in this game range from 2 to 14. Figure 5.2 ⍰ a1 a2 a3 a1 b1 b2 b3A two-ply game tree. The nodes are "MAX nodes," in which it is MAX's turn to move, and the nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is , because it leads to the state with the highest minimax value, and MIN's best reply is , because it leads to the state with the lowest minimax value. ply Given a game tree, the optimal strategy can be determined by working out the minimax value of each state in the tree, which we write as MINIMAX(s). The minimax value is the utility (for MAX) of being in that state, assuming that both players play optimally from there to the end of the game. The minimax value of a terminal state is just its utility. In a non-terminal state, MAX prefers to move to a state of maximum value when it is MAX's turn to move, and MIN prefers a state of minimum value (that is, minimum value for⎨MAX and thus maximum value for MIN). So we have: Minimax value △ ▽ a1 b1 MINIMAX (s) = ⎧ ⎪ ⎫ ⎪ ⎩ UTILITY(s,MAX) if IS-TERMINAL (s) maxa∈Actions(s) MINIMAX (RESULT (s, a)) if TO-MOVE (s) = MAX mina∈Actions(s) MINIMAX (RESULT (s, a)) if TO-MOVE (s) = MINLet us apply these definitions to the game tree in Figure 5.2 . The terminal nodes on the bottom level get their utility values from the game's UTILITY function. The first MIN node, labeled , has three successor states with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3. We can also identify the minimax decision at the root: action is the optimal choice for MAX because it leads to the state with the highest minimax value. Minimax decision This definition of optimal play for MAX assumes that MIN also plays optimally. What if MIN does not play optimally? Then MAX will do at least as well as against an optimal player, possibly better. However, that does not mean that it is always best to play the minimax optimal move when facing a suboptimal opponent. Consider a situation where optimal play by both sides will lead to a draw, but there is one risky move for MAX that leads to a state in which there are 10 possible response moves by MIN that all seem reasonable, but 9 of them are a loss for MIN and one is a loss for MAX. If MAX believes that MIN does not have sufficient computational power to discover the optimal move, MAX might want to try the risky move, on the grounds that a 9/10 chance of a win is better than a certain draw. 5.2.1 The minimax search algorithm Now that we can compute MINIMAX(s), we can turn that into a search algorithm that finds the best move for MAX by trying all actions and choosing the one whose resulting state has the highest MINIMAX value. Figure 5.3 shows the algorithm. It is a recursive algorithm that proceeds all the way down to the leaves of the tree and then backs up the minimax values through the tree as the recursion unwinds. For example, in Figure 5.2 , the algorithm first recurses down to the three bottom-left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node . A similar process gives the backed-up values of 2 for and 2 for . Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node. � B a1 � � B C DFigure 5.3 An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there. The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is and there are legal moves at each point, then the time complexity of the minimax algorithm is . The space complexity is for an algorithm that generates all actions at once, or for an algorithm that generates actions one at a time (see page 80). The exponential complexity makes MINIMAX impractical for complex games; for example, chess has a branching factor of about 35 and the average game has depth of about 80 ply, and it is not feasible to search states. MINIMAX does, however, serve as a basis for the mathematical analysis of games. By approximating the minimax analysis in various ways, we can derive more practical algorithms. m b O(b m) O(bm) O(m) 35 80 ≈ 10 1235.2.2 Optimal decisions in multiplayer games Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting new conceptual issues. First, we need to replace the single value for each node with a vector of values. For example, in a three-player game with players , , and , a vector is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities. Now we have to consider nonterminal states. Consider the node marked in the game tree shown in Figure 5.4 . In that state, player chooses what to do. The two choices lead to terminal states with utility vectors and . Since 6 is bigger than 3, should choose the first move. This means that if state is reached, subsequent play will lead to a terminal state with utilities . Hence, the backed-up value of is this vector. In general, the backed-up value of a node is the utility vector of the successor state with the highest value for the player choosing at . Figure 5.4 The first three ply of a game tree with three players (, ,). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root. Anyone who plays multiplayer games, such as Diplomacy or Settlers of Catan, quickly becomes aware that much more is going on than in two-player games. Multiplayer games A B C ⟨vA,vB,vC⟩ X ⍰ C ⟨vA = 1,vB = 2,vC = 6⟩ ⟨vA = 4,vB = 2,vC = 3⟩ C X ⟨vA = 1,vB = 2,vC = 6⟩ X n n A B Cusually involve alliances, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. How are we to understand such behavior? Are alliances a natural consequence of optimal strategies for each player in a multiplayer game? It turns out that they can be. Alliance For example, suppose and are in weak positions and is in a stronger position. Then it is often optimal for both and to attack rather than each other, lest destroy each of them individually. In this way, collaboration emerges from purely selfish behavior. Of course, as soon as weakens under the joint onslaught, the alliance loses its value, and either or could violate the agreement. In some cases, explicit alliances merely make concrete what would have happened anyway. In other cases, a social stigma attaches to breaking an alliance, so players must balance the immediate advantage of breaking an alliance against the long-term disadvantage of being perceived as untrustworthy. See Section 18.2 for more on these complications. If the game is not zero-sum, then collaboration can also occur with just two players. Suppose, for example, that there is a terminal state with utilities and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal. 5.2.3 Alpha–Beta Pruning The number of game states is exponential in the depth of the tree. No algorithm can completely eliminate the exponent, but we can sometimes cut it in half, computing the correct minimax decision without examining every state by pruning (see page 90) large parts of the tree that make no difference to the outcome. The particular technique we examine is called alpha–beta pruning. A B C A B C C C A B ⍰ ⟨vA = 1000,vB = 1000⟩Alpha–beta pruning Consider again the two-ply game tree from Figure 5.2 . Let's go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process. The steps are explained in Figure 5.5 . The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes. Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2 . At each point, we show the range of possible values for each node. (a) The first leaf below has the value 3. Hence, , which is a MIN node, has a value of at most 3. (b) The second leaf below has a value of 12; MIN would avoid this move, so the value of is still at most 3. (c) The third leaf below has a value of 8; we have seen all 's successor states, so the value of is exactly 3. Now we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root. (d) The first leaf below has the value 2. Hence, , which is a MIN node, has a value of at most 2. But we know that is worth 3, so MAX would never choose . Therefore, there is no point in looking at the other successor states of . This is an example of alpha–beta pruning. (e) The first leaf below has the value 14, so is worth at most 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring 's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second ⍰ ⍰ ⍰ B B B B B B B C C B C C D D Dsuccessor of is worth 5, so again we need to keep exploring. The third successor is worth 2, so now is worth exactly 2. MAX's decision at the root is to move to , giving a value of 3. Another way to look at this is as a simplification of the formula for MINIMAX. Let the two unevaluated successors of node in Figure 5.5 have values and . Then the value of the root node is given by In other words, the value of the root and hence the minimax decision are independent of the values of the leaves and , and therefore they can be pruned. Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node somewhere in the tree (see Figure 5.6), such that Player has a choice of moving to . If Player has a better choice either at the same level (e.g. in Figure 5.6) or at any point higher up in the tree (e.g. in Figure 5.6), then Player will never move to . So once we have found out enough about (by examining some of its descendants) to reach this conclusion, we can prune it. Figure 5.6 D D B C x y MINIMAX(root) = max(min(3,12,8), min(2,x,y), min(14,5,2)) = max(3, min(2,x,y),2) = max(3,z,2) where z = min(2,x,y) ≤ 2 = 3. x y n n m′ m n nThe general case for alpha–beta pruning. If or is better than for Player, we will never get to in play. Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha–beta pruning gets its name from the two extra parameters in MAX-VALUE (see Figure 5.7) that describe bounds on the backed-up values that appear anywhere along the path: the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX. Think: “at least.” the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN. Think: “at most.” Figure 5.7 m m′ n n (state,α,β) α = α = β = β =The alpha–beta search algorithm. Notice that these functions are the same as the MINIMAX-SEARCH functions in Figure 5.3 , except that we maintain bounds in the variables and , and use them to cut off search when a value is outside the bounds. Alpha–beta search updates the values of and as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current or value for MAX or MIN, respectively. The complete algorithm is given in Figure 5.7 . Figure 5.5 traces the progress of the algorithm on a game tree. 5.2.4 Move ordering The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. For example, in Figure 5.5(e) and (f) , we could not prune any α β α β α β   successors of at all because the worst successors (from the point of view of MIN) were generated first. If the third successor of had been generated first, with value 2, we would have been able to prune the other two successors. This suggests that it might be worthwhile to try to first examine the successors that are likely to be best. If this could be done perfectly, alpha–beta would need to examine only nodes to pick the best move, instead of for minimax. This means that the effective branching factor becomes instead of —for chess, about 6 instead of 35. Put another way, alpha–beta with perfect move ordering can solve a tree roughly twice as deep as minimax in the same amount of time. With random move ordering, the total number of nodes examined will be roughly for moderate . Now, obviously we cannot achieve perfect move ordering—in that case the ordering function could be used to play a perfect game! But we can often get fairly close. For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case result. Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit. The past could be the previous move—often the same threats remain—or it could come from previous exploration of the current move through a process of iterative deepening (see page 80). First, search one ply deep and record the ranking of moves based on their evaluations. Then search one ply deeper, using the previous ranking to inform move ordering; and so on. The increased search time from iterative deepening can be more than made up from better move ordering. The best moves are known as killer moves, and to try them first is called the killer move heuristic. Killer moves In Section 3.3.3 , we noted that redundant paths to repeated states can cause an exponential increase in search cost, and that keeping a table of previously reached states can address this problem. In game tree search, repeated states can occur because of transpositions—different permutations of the move sequence that end up in the same D D O(bm/2) O(bm) √b b O(b 3m/4) b O(bm/2) position, and the problem can be addressed with a transposition table that caches the heuristic value of states. Transposition Transposition table For example, suppose White has a move that can be answered by Black with and an unrelated move on the other side of the board that can be answered by , and that we search the sequence of moves ; let's call the resulting state . After exploring a large subtree below , we find its backed-up value, which we store in the transposition table. When we later search the sequence of moves , we end up in again, and we can look up the value instead of repeating the search. In chess, use of transposition tables is very effective, allowing us to double the reachable search depth in the same amount of time. Even with alpha–beta pruning and clever move ordering, minimax won't work for games like chess and Go, because there are still too many states to explore in the time available. In the very first paper on computer game-playing, Programming a Computer for Playing Chess (Shannon, 1950), Claude Shannon recognized this problem and proposed two strategies: a Type A strategy considers all possible moves to a certain depth in the search tree, and then uses a heuristic evaluation function to estimate the utility of states at that depth. It explores a wide but shallow portion of the tree. A Type B strategy ignores moves that look bad, and follows promising lines "as far as possible." It explores a deep but narrow portion of the tree. Type A strategy w1 b1 w2 b2 [w1,b1,w2,b2] s s [w2,b2,w1,b1] sType B strategy Historically, most chess programs have been Type A (which we cover in the next section), whereas Go programs are more often Type B (covered in Section 5.4), because the branching factor is much higher in Go. More recently, Type B programs have shown world champion-level play across a variety of games, including chess (Silver et al., 2018). 5.3 Heuristic Alpha–Beta Tree Search To make use of our limited computation time, we can cut off the search early and apply a heuristic evaluation function to states, effectively treating nonterminal nodes as if they were terminal. In other words, we replace the UTILITY function with EVAL, which estimates a state's utility. We also replace the terminal test by a cutoff test, which must return true for terminal states, but is otherwise free to decide when to cut off the search, based on the search depth and any property of the state that it chooses to consider. That gives us the formula H MINIMAX(s, d) for the heuristic minimax value of state at search depth : cutoff test 5.3.1 Evaluation functions A heuristic evaluation function returns an estimate of the expected utility of state to player , just as the heuristic functions of Chapter 3 return an estimate of the distance to the goal. For terminal states, it must be that and for nonterminal states, the evaluation must be somewhere between a loss and a win: . Beyond those requirements, what makes for a good evaluation function? First, the computation must not take too long! (The whole point is to search faster.) Second, the evaluation function should be strongly correlated with the actual chances of winning. One might well wonder about the phrase "chances of winning." After all, chess is not a game of chance: we know the current state with certainty, and no dice are involved; if neither player makes a mistake, the outcome is predetermined. But if the search must be cut off at nonterminal states, then the algorithm will necessarily be uncertain about the final outcomes s d H-MINIMAX (s, d) = ⎧⎪⎨⎪⎩ EVAL (s, MAX) if IS-CUTOFF (s, d) maxa∈Actions(s) H-MINIMAX (RESULT (s, a), d + 1) if TO-MOVE (s) = MAX mina∈Actions(s) H-MINIMAX (RESULT (s, a), d + 1) if TO-MOVE (s) = MIN. EVAL(s, p) s p EVAL(s, p) = UTILITY(s, p) UTILITY(loss, p) ≤ EVAL(s, p) ≤ UTILITY(win, p)of those states (even though that uncertainty could be resolved with infinite computing resources). Let us make this idea more concrete. Most evaluation functions work by calculating various features of the state—for example, in chess, we would have features for the number of white pawns, black pawns, white queens, black queens, and so on. The features, taken together, define various categories or equivalence classes of states: the states in each category have the same values for all the features. For example, one category might contain all two-pawn versus one-pawn endgames. Any given category will contain some states that lead (with perfect play) to wins, some that lead to draws, and some that lead to losses. Features The evaluation function does not know which states are which, but it can return a single value that estimates the proportion of states with each outcome. For example, suppose our experience suggests that 82% of the states encountered in the two-pawns versus one-pawn category lead to a win (utility); 2% to a loss (0), and 16% to a draw . Then a reasonable evaluation for states in the category is the expected value: . In principle, the expected value can be determined for each category of states, resulting in an evaluation function that works for any state. Expected value In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities. Instead, most evaluation functions compute separate numerical contributions from each feature and then combine them to find the total value. For centuries, chess players have developed ways of judging the value of a position using just this idea. For example, introductory chess books give an approximate material +1 (1/2) (0.82 × +1) + (0.02 × 0) + (0.16 × 1/2) = 0.90value for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9. Other features such as “good pawn structure” and “king safety” might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position. Material value Mathematically, this kind of evaluation function is called a weighted linear function because it can be expressed as Weighted linear function where each is a feature of the position (such as “number of white bishops”) and each is a weight (saying how important that feature is). The weights should be normalized so that the sum is always within the range of a loss (0) to a win . A secure advantage equivalent to a pawn gives a substantial likelihood of winning, and a secure advantage equivalent to three pawns should give almost certain victory, as illustrated in Figure 5.8(a) . We said that the evaluation function should be strongly correlated with the actual chances of winning, but it need not be linearly correlated: if state is twice as likely to win as state we don’t require that EVAL(S) be twice EVAL(S'); all we require is that . Figure 5.8 EVAL(s) = w1f1(s) + w2f2(s) + ⋯ + wnfn(s) = n∑ i=1 wifi(s), fi wi (+1) ⍰ s s ′ EVAL(s) > EVAL(s')Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win. Adding up the values of features seems like a reasonable thing to do, but in fact it involves a strong assumption: that the contribution of each feature is independent of the values of the other features. For this reason, current programs for chess and other games also use nonlinear combinations of features. For example, a pair of bishops might be worth more than twice the value of a single bishop, and a bishop is worth more in the endgame than earlier—when the move number feature is high or the number of remaining pieces feature is low. Where do the features and weights come from? They’re not part of the rules of chess, but they are part of the culture of human chess-playing experience. In games where this kind of experience is not available, the weights of the evaluation function can be estimated by the machine learning techniques of Chapter 22 . Applying these techniques to chess has confirmed that a bishop is indeed worth about three pawns, and it appears that centuries of human experience can be replicated in just a few hours of machine learning. 5.3.2 Cutting off search The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search. We replace the two lines in Figure 5.7 that mention IS-TERMINAL with the following line: if game.IS-CUTOFF(state, depth) then return game.EVAL(state, player), null ⍰ ⍰We also must arrange for some bookkeeping so that the current depth is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to set a fixed depth limit so that IS-CUTOFF(state, depth) returns true for all depth greater than some fixed depth (as well as for all terminal states). The depth is chosen so that a move is selected within the allocated time. A more robust approach is to apply iterative deepening. (See Chapter 3 .) When time runs out, the program returns the move selected by the deepest completed search. As a bonus, if in each round of iterative deepening we keep entries in the transposition table, subsequent rounds will be faster, and we can use the evaluations to improve move ordering. These simple approaches can lead to errors due to the approximate nature of the evaluation function. Consider again the simple evaluation function for chess based on material advantage. Suppose the program searches to the depth limit, reaching the position in Figure 5.8(b) , where Black is ahead by a knight and two pawns.

It would report this as the heuristic value of the state, thereby declaring that the state is a probable win by Black. But White's next move captures Black's queen with no compensation. Hence, the position is actually favorable for White, but this can be seen only by looking ahead. The evaluation function should be applied only to positions that are quiescent—that is, positions in which there is no pending move (such as a capturing the queen) that would wildly swing the evaluation. For nonquiescent positions the IS-CUTOFF returns false, and the search continues until quiescent positions are reached. This extra quiescence search is sometimes restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position. Quiescence Quiescence search d d The horizon effect is more difficult to eliminate. It arises when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by the use of delaying tactics. Consider the chess position in Figure 5.9 . It is clear that there is no way for the black bishop to escape. For example, the white rook can capture it by moving to h1, then a1, then a2; a capture at depth 6 ply. Figure 5.9 The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, encouraging the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones. Horizon effect But Black does have a sequence of moves that pushes the capture of the bishop "over the horizon." Suppose Black searches to depth 8 ply. Most moves by Black will lead to the eventual capture of the bishop, and thus will be marked as "bad" moves. But Black will also consider the sequence of moves that starts by checking the king with a pawn, and enticing the king to capture the pawn. Black can then do the same thing with a second pawn. That takes up enough moves that the capture of the bishop would not be discovered during the remainder of Black's search. Black thinks that the line of play has saved the bishop at the price of two pawns, when actually all it has done is waste pawns and push the inevitable capture of the bishop beyond the horizon that Black can see. One strategy to mitigate the horizon effect is to allow singular extensions, moves that are "clearly better" than all other moves in a given position, even when the search would normally be cut off at that point. In our example, a search will have revealed that three moves of the white rook—h2 to h1, then h1 to a1, and then a1 capturing the bishop on a2— are each in turn clearly better moves, so even if a sequence of pawn moves pushes us to the horizon, these clearly better moves will be given a chance to extend the search. This makes the tree deeper, but because there are usually few singular extensions, the strategy does not add many total nodes to the tree, and has proven to be effective in practice. Singular extension 5.3.3 Forward pruning Alpha–beta pruning prunes branches of the tree that can have no effect on the final evaluation, but forward pruning prunes moves that appear to be poor moves, but might possibly be good ones. Thus, the strategy saves computation time at the risk of making an error. In Shannon's terms, this is a Type B strategy. Clearly, most human chess players do this, considering only a few moves from each position (at least consciously). Forward pruning One approach to forward pruning is beam search (see page 115): on each ply, consider only a "beam" of the n best moves (according to the evaluation function) rather than consideringall possible moves. Unfortunately, this approach is rather dangerous because there is no guarantee that the best move will not be pruned away. The PROBCUT, or probabilistic cut, algorithm (Buro, 1995) is a forward-pruning version of alpha–beta search that uses statistics gained from prior experience to lessen the chance that the best move will be pruned. Alpha–beta search prunes any node that is provably outside the current window. PROBCUT also prunes nodes that are probably outside the window. It computes this probability by doing a shallow search to compute the backed-up value of a node and then using past experience to estimate how likely it is that a score of at depth in the tree would be outside . Buro applied this technique to his Othello program, LOGISTELLO, and found that a version of his program with PROBCUT beat the regular version 64% of the time, even when the regular version was given twice as much time. Another technique, late move reduction, works under the assumption that move ordering has been done well, and therefore moves that appear later in the list of possible moves are less likely to be good moves. But rather than pruning them away completely, we just reduce the depth to which we search these moves, thereby saving time. If the reduced search comes back with a value above the current value, we can re-run the search with the full depth. Late move reduction Combining all the techniques described here results in a program that can play creditable chess (or other games). Let us assume we have implemented an evaluation function for chess, a reasonable cutoff test with a quiescence search. Let us also assume that, after months of tedious bit-bashing, we can generate and evaluate around a million nodes per second on the latest PC. The branching factor for chess is about 35, on average, and is about 50 million, so if we used minimax search, we could look ahead only five ply in about a minute of computation; the rules of competition would not give us enough time to search six ply. Though not incompetent, such a program can be defeated by an average human chess player, who can occasionally plan six or eight ply ahead. (α, β) v v d (α, β) α 35 5With alpha–beta search and a large transposition table we get to about 14 ply, which results in an expert level of play. We could trade in our PC for a workstation with 8 GPUs, getting us over a billion nodes per second, but to obtain grandmaster status we would still need an extensively tuned evaluation function and a large database of endgame moves. Top chess programs like STOCKFISH have all of these, often reaching depth 30 or more in the search tree and far exceeding the ability of any human player. 5.3.4 Search versus lookup Somehow it seems like overkill for a chess program to start a game by considering a tree of a billion game states, only to conclude that it will play pawn to e4 (the most popular first move). Books describing good play in the opening and endgame in chess have been available for more than a century (Tattersall, 1911). It is not surprising, therefore, that many game-playing programs use table lookup rather than search for the opening and ending of games. For the openings, the computer is mostly relying on the expertise of humans. The best advice of human experts on how to play each opening can be copied from books and entered into tables for the computer’s use. In addition, computers can gather statistics from a database of previously played games to see which opening sequences most often lead to a win. For the first few moves there are few possibilities, and most positions will be in the table. Usually after about 10 or 15 moves we end up in a rarely seen position, and the program must switch from table lookup to search. Near the end of the game there are again fewer possible positions, and thus it is easier to do lookup. But here it is the computer that has the expertise: computer analysis of endgames goes far beyond human abilities. Novice humans can win a king-and-rook-versus-king (KRK) endgame by following a few simple rules. Other endings, such as king, bishop, and knight versus king (KBNK), are difficult to master and have no succinct strategy description. A computer, on the other hand, can completely solve the endgame by producing a policy, which is a mapping from every possible state to the best move in that state. Then the computer can play perfectly by looking up the right move in this table. The table is constructed by retrograde minimax search: start by considering all ways to place the KBNK pieces on the board. Some of the positions are wins for white; mark them as such. Then reverse the rules of chess to do reverse moves rather than moves. Any move by White that,no matter what move Black responds with, ends up in a position marked as a win, must also be a win. Continue this search until all possible positions are resolved as win, loss, or draw, and you have an infallible lookup table for all endgames with those pieces. This has been done not only for KBNK endings, but for all endings with seven or fewer pieces. The tables contain 400 trillion positions. An eight-piece table would require 40 quadrillion positions. Retrograde5.4 Monte Carlo Tree Search The game of Go illustrates two major weaknesses of heuristic alpha–beta tree search: First, Go has a branching factor that starts at 361, which means alpha–beta search would be limited to only 4 or 5 ply. Second, it is difficult to define a good evaluation function for Go because material value is not a strong indicator and most positions are in flux until the endgame. In response to these two challenges, modern Go programs have abandoned alpha–beta search and instead use a strategy called Monte Carlo tree search (MCTS). 3 “Monte Carlo” algorithms are randomized algorithms named after the Casino de Monte-Carlo in Monaco. Monte Carlo tree search (MCTS) The basic MCTS strategy does not use a heuristic evaluation function. Instead, the value of a state is estimated as the average utility over a number of simulations of complete games starting from the state. A simulation (also called a playout or rollout) chooses moves first for one player, than for the other, repeating until a terminal position is reached. At that point the rules of the game (not fallible heuristics) determine who has won or lost, and by what score. For games in which the only outcomes are a win or a loss, “average utility” is the same as “win percentage.” Simulation Playout 3Rollout How do we choose what moves to make during the playout? If we just choose randomly, then after multiple simulations we get an answer to the question “what is the best move if both players play randomly?” For some simple games, that happens to be the same answer as “what is the best move if both players play well?,” but for most games it is not. To get useful information from the playout we need a playout policy that biases the moves towards good ones. For Go and other games, playout policies have been successfully learned from self-play by using neural networks. Sometimes game-specific heuristics are used, such as “consider capture moves” in chess or “take the corner square” in Othello. Playout policy Given a playout policy, we next need to decide two things: from what positions do we start the playouts, and how many playouts do we allocate to each position? The simplest answer, called pure Monte Carlo search, is to do simulations starting from the current state of the game, and track which of the possible moves from the current position has the highest win percentage. Pure Monte Carlo search For some stochastic games this converges to optimal play as increases, but for most games it is not sufficient—we need a selection policy that selectively focuses the computational resources on the important parts of the game tree. It balances two factors: exploration of states that have had few playouts, and exploitation of states that have done well in past playouts, to get a more accurate estimate of their value. (See Section 17.3 for more on the exploration/exploitation tradeoff.) Monte Carlo tree search does that by N N ⍰maintaining a search tree and growing it on each iteration of the following four steps, as shown in Figure 5.10 : Selection policy Exploration Exploitation SELECTION: Starting at the root of the search tree, we choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf. Figure 5.10(a) shows a search tree with the root representing a state where white has just moved, and white has won 37 out of the 100 playouts done so far. The thick arrow shows the selection of a move by black that leads to a node where black has won 60/79 playouts. This is the best win percentage among the three moves, so selecting it is an example of exploitation. But it would also have been reasonable to select the 2/11 node for the sake of exploration—with only 11 playouts, the node still has high uncertainty in its valuation, and might end up being best if we gain more information about it. Selection continues on to the leaf node marked 27/35. EXPANSION: We grow the search tree by generating a new child of the selected node; Figure 5.10(b) shows the new node marked with 0/0. (Some versions generate more than one child in this step.) SIMULATION: We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are not recorded in the search tree. In the figure, the simulation results in a win for black. BACK-PROPAGATION: We now use the result of the simulation to update all the search tree nodes going up to the root. Since black won the playout, black nodes are incremented in both the number of wins and the number of playouts, so 27/35 becomes ⍰ ⍰ ⍰28/26 and 60/79 becomes 61/80. Since white lost, the white nodes are incremented in the number of playouts only, so 16/53 becomes 16/54 and the root 37/100 becomes 37/101. Figure 5.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked (for 27 wins for black out of 35 playouts). In (b) we expand the selected node and do a simulation (playout), which ends in a win for black. In (c), the results of the simulation are back-propagated up the tree. We repeat these four steps either for a set number of iterations, or until the allotted time has expired, and then return the move with the highest number of playouts. One very effective selection policy is called “upper confidence bounds applied to trees” or UCT. The policy ranks each possible move based on an upper confidence bound formula called UCB1. (See Section 17.3.3 for more details.) For a node , the formula is: UCT (27/35) ⍰ n UCB1(n) = + C × √ U(n) N(n) log N(PARENT(n)) N(n)UCB1 where is the total utility of all playouts that went through node , is the number of playouts through node , and is the parent node of in the tree. Thus is the exploitation term: the average utility of . The term with the square root is the exploration term: it has the count in the denominator, which means the term will be high for nodes that have only been explored a few times. In the numerator it has the log of the number of times we have explored the parent of . This means that if we are selecting some non-zero percentage of the time, the exploration term goes to zero as the counts increase, and eventually the playouts are given to the node with highest average utility. is a constant that balances exploitation and exploration. There is a theoretical argument that should be , but in practice, game programmers try multiple values for and choose the one that performs best. (Some programs use slightly different formulas; for example, ALPHAZERO adds in a term for move probability, which is calculated by a neural network trained from past self-play.) With , the 60/79 node in Figure 5.10 has the highest UCB1 score, but with , it would be the node. Figure 5.11 shows the complete UCT MCTS algorithm. When the iterations terminate, the move with the highest number of playouts is returned. You might think that it would be better to return the node with the highest average utility, but the idea is that a node with wins is better than one with wins, because the latter has a lot of uncertainty. In any event, the UCB1 formula ensures that the node with the most playouts is almost always the node with the highest win percentage, because the selection process favors win percentage more and more as the number of playouts goes up. Figure 5.11 U(n) n N(n) n PARENT(n) n U(n) N(n) n N(n) n n C C √2 C C = 1.4 ⍰ C = 1.5 2/11 ⍰ 65/100 2/3The Monte Carlo tree search algorithm. A game tree, tree, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts. The time to compute a playout is linear, not exponential, in the depth of the game tree, because only one move is taken at each choice point. That gives us plenty of time for multiple playouts. For example: consider a game with a branching factor of 32, where the average game lasts 100 ply. If we have enough computing power to consider a billion game states before we have to make a move, then minimax can search 6 ply deep, alpha–beta with perfect move ordering can search 12 ply, and Monte Carlo search can do 10 million playouts. Which approach will be better? That depends on the accuracy of the heuristic function versus the selection and playout policies. The conventional wisdom has been that Monte Carlo search has an advantage over alpha–beta for games like Go where the branching factor is very high (and thus alpha–beta can’t search deep enough), or when it is difficult to define a good evaluation function. What alpha–beta does is choose the path to a node that has the highest achievable evaluation function score, given that the opponent will be trying to minimize the score. Thus, if the evaluation function is inaccurate, alpha–beta will be inaccurate. A miscalculation on a single node can lead alpha–beta to erroneously choose (or avoid) a path to that node. But Monte Carlo search relies on the aggregate of many playouts, and thus is not as vulnerable to a single error. It is possible to combine MCTS and evaluation functions by doing a playout for a certain number of moves, but then truncating the playout and applying an evaluation function. It is also possible to combine aspects of alpha–beta and Monte Carlo search. For example, in games that can last many moves, we may want to use early playout termination, in which we stop a playout that is taking too many moves, and either evaluate it with a heuristic evaluation function or just declare it a draw.Early playout termination Monte Carlo search can be applied to brand-new games, in which there is no body of experience to draw upon to define an evaluation function. As long as we know the rules of the game, Monte Carlo search does not need any additional information. The selection and playout policies can make good use of hand-crafted expert knowledge when it is available, but good policies can be learned using neural networks trained by self-play alone. Monte Carlo search has a disadvantage when it is likely that a single move can change the course of the game, because the stochastic nature of Monte Carlo search means it might fail to consider that move. In other words, Type B pruning in Monte Carlo search means that a vital line of play might not be explored at all. Monte Carlo search also has a disadvantage when there are game states that are “obviously” a win for one side or the other (according to human knowledge and to an evaluation function), but where it will still take many moves in a playout to verify the winner. It was long held that alpha–beta search was better suited for games like chess with low branching factor and good evaluation functions, but recently Monte Carlo approaches have demonstrated success in chess and other games. The general idea of simulating moves into the future, observing the outcome, and using the outcome to determine which moves are good ones is one kind of reinforcement learning, which is covered in Chapter 22 �.5.5 Stochastic Games Stochastic games bring us a little closer to the unpredictability of real life by including a random element, such as the throwing of dice. Backgammon is a typical stochastic game that combines luck and skill. In the backgammon position of Figure 5.12 , White has rolled a 6–5 and has four possible moves (each of which moves one piece forward (clockwise) 5 positions, and one piece forward 6 positions). Figure 5.12 A typical backgammon position. The goal of the game is to move all one’s pieces off the board. Black moves clockwise toward 25, and White moves counterclockwise toward 0. A piece can move to any �position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, Black has rolled and must choose among four legal moves: where the notation means move one piece from position 5 to 11, and then move a piece from 11 to 16. Stochastic game At this point Black knows what moves can be made, but does not know what White is going to roll and thus does not know what White’s legal moves will be. That means Black cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe. A game tree in backgammon must include chance nodes in addition to MAX and MIN nodes. Chance nodes are shown as circles in Figure 5.13 . The branches leading from each chance node denote the possible dice rolls; each branch is labeled with the roll and its probability. There are 36 ways to roll two dice, each equally likely; but because a 6–5 is the same as a 5–6, there are only 21 distinct rolls. The six doubles (1–1 through 6–6) each have a probability of 1/36, so we say . The other 15 distinct rolls each have a 1/18 probability. Figure 5.13 $6 - 5$ $(5 - 11, 5 - 10)$, $(5 - 11, 19 - 24)$, $(5 - 10, 10 - 16)$, and $(5 - 11, 11 - 16)$, $(5 - 11, 11 - 16)$ � $P(1–1) = 1/36$Schematic game tree for a backgammon position. Chance nodes The next step is to understand how to make correct decisions. Obviously, we still want to pick the move that leads to the best position. However, positions do not have definite minimax values. Instead, we can only calculate the expected value of a position: the average over all possible outcomes of the chance nodes. Expected valueThis leads us to the expectiminimax value for games with chance nodes, a generalization of the minimax value for deterministic games. Terminal nodes and MAX and MIN nodes work exactly the same way as before (with the caveat that the legal moves for MAX and MIN will depend on the outcome of the dice roll in the previous chance node). For chance nodes we compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action: Expectiminimax value where represents a possible dice roll (or other chance event) and is the same state as , with the additional fact that the result of the dice roll is . 5.5.1 Evaluation functions for games of chance As with minimax, the obvious approximation to make with expectiminimax is to cut the search off at some point and apply an evaluation function to each leaf. One might think that evaluation functions for games such as backgammon should be just like evaluation functions for chess—they just need to give higher values to better positions. But in fact, the presence of chance nodes means that one has to be more careful about what the values mean. Figure 5.14 shows what happens: with an evaluation function that assigns the values [1, 2, 3, 4] to the leaves, move is best; with values [1, 20, 30, 400], move is best. Hence, the program behaves totally differently if we make a change to some of the evaluation values, even if the preference order remains the same. Figure 5.14 EXPECTIMINIMAX(s) = ⎧ ⎪ ⎪ ⎪ ⎪ ⎪ ⎪ ⎨ ⎪ ⎪ ⎪ ⎪ ⎪ ⎪ ⎩ UTILITY(s,MAX) if IS-TERMINAL(s) maxaif EXPECTIMINIMAX(RESULT(s, a)) if To-Move(s) = MAX minaif EXPECTIMINIMAX(RESULT(s, a)) if To-Move(s) = MIN ∑rP(r)EXPECTIMINIMAX(RESULT(s, r)) if To-Move(s) = CHANCE r RESULT(s, r) s r □ a1 a2An order-preserving transformation on leaf values changes the best move. It turns out that to avoid this problem, the evaluation function must return values that are a positive linear transformation of the probability of winning (or of the expected utility, for games that have outcomes other than win/lose). This relation to probability is an important and general property of situations in which uncertainty is involved, and we discuss it further in Chapter 16 . If the program knew in advance all the dice rolls that would occur for the rest of the game, solving a game with dice would be just like solving a game without dice, which minimax does in time, where is the branching factor and is the maximum depth of the game tree. Because expectiminimax is also considering all the possible dice-roll sequences, it will take , where is the number of distinct rolls. Even if the search is limited to some small depth , the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far in most games of chance. In backgammon is 21 and is usually around 20, but in some situations can be as high as 4000 for dice rolls that are doubles. We could probably only manage three ply of search. Another way to think about the problem is this: the advantage of alpha–beta is that it ignores future developments that just are not going to happen, given best play. Thus, it concentrates on likely occurrences. But in a game where a throw of two dice precedes each move, there are no likely sequences of moves; even the most likely move occurs only of the time, because for the move to take place, the dice would first have to come out the right way to make it legal. This is a general problem whenever uncertainty enters the picture: the □ O(bm) b m O(bmnm) n d n b 1/2possibilities are multiplied enormously, and forming detailed plans of action becomes pointless because the world probably will not play along. It may have occurred to you that something like alpha–beta pruning could be applied to game trees with chance nodes. It turns out that it can. The analysis for MIN and MAX nodes is unchanged, but we can also prune chance nodes, using a bit of ingenuity. Consider the chance node in Figure 5.13 and what happens to its value as we evaluate its children. Is it possible to find an upper bound on the value of before we have looked at all its children? (Recall that this is what alpha–beta needs in order to prune a node and its subtree.) At first sight, it might seem impossible because the value of is the average of its children's values, and in order to compute the average of a set of numbers, we must look at all the numbers. But if we put bounds on the possible values of the utility function, then we can arrive at bounds for the average without looking at every number. For example, say that all utility values are between and ; then the value of leaf nodes is bounded, and in turn we can place an upper bound on the value of a chance node without looking at all its children. In games where the branching factor for chance nodes is high—consider a game like Yahtzee where you roll 5 dice on every turn—you may want to consider forward pruning that samples a smaller number of the possible chance branches. Or you may want to avoid using an evaluation function altogether, and opt for Monte Carlo tree search instead, where each playout includes random dice rolls. C □ C C −2 +25.6 Partially Observable Games Bobby Fischer declared that "chess is war," but chess lacks at least one major characteristic of real wars, namely, partial observability. In the "fog of war," the whereabouts of enemy units is often unknown until revealed by direct contact. As a result, warfare includes the use of scouts and spies to gather information and the use of concealment and bluff to confuse the enemy. Partially observable games share these characteristics and are thus qualitatively different from the games in the preceding sections. Video games such as StarCraft are particularly challenging, being partially observable, multi-agent, nondeterministic, dynamic, and unknown. In deterministic partially observable games, uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent. This class includes children's games such as Battleship (where each player's ships are placed in locations hidden from the opponent) and Stratego (where piece locations are known but piece types are hidden). We will examine the game of Kriegspiel, a partially observable variant of chess in which pieces are completely invisible to the opponent. Other games also have partially observable versions: Phantom Go, Phantom tic-tac-toe, and Screen Shogi. Kriegspiel 5.6.1 Kriegspiel: Partially observable chess The rules of Kriegspiel are as follows: White and Black each see a board containing only their own pieces. A referee, who can see all the pieces, adjudicates the game and periodically makes announcements that are heard by both players. First, White proposes to the referee a move that would be legal if there were no black pieces. If the black pieces prevent the move, the referee announces "illegal," and White keeps proposing moves until a legal one is found—learning more about the location of Black's pieces in the process.Once a legal move is proposed, the referee announces one or more of the following: "Capture on square X" if there is a capture, and "Check by D" if the black king is in check, where D is the direction of the check, and can be one of "Knight," "Rank," "File," "Long diagonal," or "Short diagonal." If Black is checkmated or stalemated, the referee says so; otherwise, it is Black's turn to move. Kriegspiel may seem terrifyingly impossible, but humans manage it quite well and computer programs are beginning to catch up. It helps to recall the notion of a belief state as defined in Section 4.4 and illustrated in Figure 4.14 —the set of all logically possible board states given the complete history of percepts to date. Initially, White's belief state is a singleton because Black's pieces haven't moved yet. After White makes a move and Black responds, White's belief state contains 20 positions, because Black has 20 replies to any opening move. Keeping track of the belief state as the game progresses is exactly the problem of state estimation, for which the update step is given in Equation (4.6) on Page 132. We can map Kriegspiel state estimation directly onto the partially observable, nondeterministic framework of Section 4.4 if we consider the opponent as the source of nondeterminism; that is, the RESULTS of White's move are composed from the (predictable) outcome of White's own move and the unpredictable outcome given by Black's reply. 4 Sometimes, the belief state will become too large to represent just as a list of board states, but we will ignore this issue for now; Chapters 7 and 8 suggest methods for compactly representing very large belief states. Given a current belief state, White may ask, "Can I win the game?" For a partially observable game, the notion of a strategy is altered; instead of specifying a move to make for each possible move the opponent might make, we need a move for every possible percept sequence that might be received. For Kriegspiel, a winning strategy, or guaranteed checkmate, is one that, for each possible percept sequence, leads to an actual checkmate for every possible board state in the current belief state, regardless of how the opponent moves. With this definition, the opponent's belief state is irrelevant—the strategy has to work even if the opponent can see all the pieces. This greatly simplifies the computation. Figure 5.15 shows part of a guaranteed checkmate for the KRK (king and rook versus king) endgame. In this case, Black has just one piece (the king), so a belief state for White can be shown in a single board by marking each possible position of the Black king. � � � � 4 � � �Figure 5.15 Part of a guaranteed checkmate in the KRK endgame, shown on a reduced board. In the initial belief state, Black's king is in one of three possible locations. By a combination of probing moves, the strategy narrows this down to one. Completion of the checkmate is left as an exercise. Guaranteed checkmate The general AND-OR search algorithm can be applied to the belief-state space to find guaranteed checkmates, just as in Section 4.4 �. The incremental belief-state algorithmmentioned in Section 4.4.2 often finds midgame checkmates up to depth 9—well beyond the abilities of most human players. In addition to guaranteed checkmates, Kriegspiel admits an entirely new concept that makes no sense in fully observable games: probabilistic checkmate. Such checkmates are still required to work in every board state in the belief state; they are probabilistic with respect to randomization of the winning player's moves. To get the basic idea, consider the problem of finding a lone black king using just the white king. Simply by moving randomly, the white king will eventually bump into the black king even if the latter tries to avoid this fate, since Black cannot keep guessing the right evasive moves indefinitely. In the terminology of probability theory, detection occurs with probability 1. Probabilistic checkmate The KBNK endgame—king, bishop and knight versus king—is won in this sense; White presents Black with an infinite random sequence of choices, for one of which Black will guess incorrectly and reveal his position, leading to checkmate. On the other hand, the KBBK endgame is won with probability . White can force a win only by leaving one of his bishops unprotected for one move. If Black happens to be in the right place and captures the bishop (a move that would be illegal if the bishops are protected), the game is drawn. White can choose to make the risky move at some randomly chosen point in the middle of a very long sequence, thus reducing to an arbitrarily small constant, but cannot reduce to zero. Sometimes a checkmate strategy works for some of the board states in the current belief state but not others. Trying such a strategy may succeed, leading to an accidental checkmate—accidental in the sense that White could not know that it would be checkmate— if Black's pieces happen to be in the right places. (Most checkmates in games between humans are of this accidental nature.) This idea leads naturally to the question of how likely it is that a given strategy will win, which leads in turn to the question of how likely it is that each board state in the current belief state is the true board state. 1 − ϵ ϵ ϵAccidental checkmate One's first inclination might be to propose that all board states in the current belief state are equally likely—but this can't be right. Consider, for example, White's belief state after Black's first move of the game. By definition (assuming that Black plays optimally), Black must have played an optimal move, so all board states resulting from suboptimal moves ought to be assigned zero probability. This argument is not quite right either, because each player's goal is not just to move pieces to the right squares but also to minimize the information that the opponent has about their location. Playing any predictable "optimal" strategy provides the opponent with information. Hence, optimal play in partially observable games requires a willingness to play somewhat randomly. (This is why restaurant hygiene inspectors do random inspection visits.) This means occasionally selecting moves that may seem "intrinsically" weak—but they gain strength from their very unpredictability, because the opponent is unlikely to have prepared any defense against them. From these considerations, it seems that the probabilities associated with the board states in the current belief state can only be calculated given an optimal randomized strategy; in turn, computing that strategy seems to require knowing the probabilities of the various states the board might be in. This conundrum can be resolved by adopting the game-theoretic notion of an equilibrium solution, which we pursue further in Chapter 17 . An equilibrium specifies an optimal randomized strategy for each player. Computing equilibria is too expensive for Kriegspiel. At present, the design of effective algorithms for general Kriegspiel play is an open research topic. Most systems perform bounded-depth look-ahead in their own belief-state space, ignoring the opponent's belief state. Evaluation functions resemble those for the observable game but include a component for the size of the belief state— smaller is better! We will return to partially observable games under the topic of Game Theory in Section 18.2 . 5.6.2 Card games Card games such as bridge, whist, hearts, and poker feature stochastic partial observability, where the missing information is generated by the random dealing of cards. At first sight, it might seem that these card games are just like dice games: the cards are dealt randomly and determine the moves available to each player, but all the "dice" are rolled at the beginning! Even though this analogy turns out to be incorrect, it suggests an algorithm: treat the start of the game as a chance node with every possible deal as an outcome, and then use the EXPECTIMINIMAX formula to pick the best move. Note that in this approach the only chance node is the root node; after that the game becomes fully observable. This approach is sometimes called averaging over clairvoyance because it assumes that once the actual deal has occurred, the game becomes fully observable to both players. Despite its intuitive appeal, the strategy can lead one astray. Consider the following story: DAY 1: Road A leads to a pot of gold; Road B leads to a fork. You can see that the left fork leads to two pots of gold, and the right fork leads to you being run over by a bus. DAY 2: Road A leads to a pot of gold; Road B leads to a fork. You can see that the right fork leads to two pots of gold, and the left fork leads to you being run over by a bus. DAY 3: Road A leads to a pot of gold; Road B leads to a fork. You are told that one fork leads to two pots of gold, and one fork leads to you being run over by a bus. Unfortunately you don’t know which fork is which. Averaging over clairvoyance leads to the following reasoning: on Day 1, B is the right choice; on Day 2, B is the right choice; on Day 3, the situation is the same as either Day 1 or Day 2, so B must still be the right choice. Now we can see how averaging over clairvoyance fails: it does not consider the belief state that the agent will be in after acting. A belief state of total ignorance is not desirable, especially when one possibility is certain death. Because it assumes that every future state will automatically be one of perfect knowledge, the clairvoyance approach never selects actions that gather information (like the first move in Figure 5.15); nor will it choose actions that hide information from the opponent or provide information to a partner, because it assumes that they already know the information; and it will never bluff in poker, because it assumes the opponent can see its cards. In Chapter 17 , we show how to construct algorithms that do all these things by virtue of solving the true partially observable decision problem, resulting in an optimal equilibrium strategy (see Section 18.2). 5 Bluffing—betting as if one’s hand is good, even when it’s not—is a core part of poker strategy. � 5 � �Bluff Despite the drawbacks, averaging over clairvoyance can be an effective strategy, with some tricks to make it work better. In most card games, the number of possible deals is rather large. For example, in bridge play, each player sees just two of the four hands; there are two unseen hands of 13 cards each, so the number of deals is Solving even one deal is quite difficult, so solving ten million is out of the question. One way to deal with this huge number is with abstraction: i.e. by treating similar hands as identical. For example, it is very important which aces and kings are in a hand, but whether the hand has a 4 or 5 is not as important, and can be abstracted away. Another way to deal with the large number is forward pruning: consider only a small random sample of deals, and again calculate the EXPECTIMINIMAX score. Even for fairly small —say, 100 to 1,000—this method gives a good approximation. It can also be applied to deterministic games such as Kriegspiel, where we sample over possible states of the game rather than over possible deals, as long as we have some way to estimate how likely each state is. It can also be helpful to do heuristic search with a depth cutoff rather than to search the entire game tree. So far we have assumed that each deal is equally likely. That makes sense for games like whist and hearts. But for bridge, play is preceded by a bidding phase in which each team indicates how many tricks it expects to win. Since players bid based on the cards they hold, the other players learn something about the probability of each deal. Taking this into account in deciding how to play the hand is tricky, for the reasons mentioned in our description of Kriegspiel: players may bid in such a way as to minimize the information conveyed to their opponents. Computers have reached a superhuman level of performance in poker. The poker program Libratus took on four of the top poker players in the world in a 20-day match of no-limit Texas hold ’em and decisively beat them all. Since there are so many possible states in poker, Libratus uses abstraction to reduce the state space: it might consider the two hands AAA72 and AAA64 to be equivalent (they’re both “three aces and some low cards”), and it might consider a bet of 200 dollars to be the same as 201 dollars. But Libratus also monitors (26 13) = 10,400,600. N N P(s)the other players, and if it detects they are exploiting an abstraction, it will do some additional computation overnight to plug that hole. Overall it used 25 million CPU hours on a supercomputer to pull off the win. The computational costs incurred by Libratus (and similar costs by ALPHAZERO and other systems) suggests that world champion game play may not be achievable for researchers with limited budgets. To some extent that is true: just as you should not expect to be able to assemble a champion Formula One race car out of spare parts in your garage, there is an advantage to having access to supercomputers or specialty hardware such as Tensor Processing Units. That is particularly true when training a system, but training could also be done via crowdsourcing. For example the open-source LEELAZERO system is a reimplementation of ALPHAZERO that trains through self-play on the computers of volunteer participants. Once trained, the computational requirements for actual tournament play are modest. ALPHASTAR won StarCraft II games running on a commodity desktop with a single GPU, and ALPHAZERO could have been run in that mode.5.7 Limitations of Game Search Algorithms Because calculating optimal decisions in complex games is intractable, all algorithms must make some assumptions and approximations. Alpha–beta search uses the heuristic evaluation function as an approximation, and Monte Carlo search computes an approximate average over a random selection of playouts. The choice of which algorithm to use depends in part on the features of each game: when the branching factor is high or it is difficult to define an evaluation function, Monte Carlo search is preferred. But both algorithms suffer from fundamental limitations. One limitation of alpha–beta search is its vulnerability to errors in the heuristic function. Figure 5.16 shows a two-ply game tree for which minimax suggests taking the right-hand branch because . That is the correct move if the evaluations are all exactly accurate. But suppose that the evaluation of each node has an error that is independent of other nodes and is randomly distributed with a standard deviation of . Then the left-hand branch is actually better 71% of the time when , and 58% of the time when (because one of the four right-hand leaves is likely to slip below 99 in these cases). If errors in the evaluation function are not independent, then the chance of a mistake rises. It is difficult to compensate for this because we don’t have a good model of the dependencies between the values of sibling nodes. Figure 5.16 A two-ply game tree for which heuristic minimax may make an error. ⍰ 100 > 99 σ σ = 5 σ = 2A second limitation of both alpha–beta and Monte Carlo is that they are designed to calculate (bounds on) the values of legal moves. But sometimes there is one move that is obviously best (for example when there is only one legal move), and in that case, there is no point wasting computation time to figure out the value of the move—it is better to just make the move. A better search algorithm would use the idea of the utility of a node expansion, selecting node expansions of high utility—that is, ones that are likely to lead to the discovery of a significantly better move. If there are no node expansions whose utility is higher than their cost (in terms of time), then the algorithm should stop searching and make a move. This works not only for clear-favorite situations but also for the case of symmetrical moves, for which no amount of search will show that one move is better than another. This kind of reasoning about what computations to do is called metareasoning (reasoning about reasoning). It applies not just to game playing but to any kind of reasoning at all. All computations are done in the service of trying to reach better decisions, all have costs, and all have some likelihood of resulting in a certain improvement in decision quality. Monte Carlo search does attempt to do metareasoning to allocate resources to the most important parts of the tree, but does not do so in an optimal way. Metareasoning A third limitation is that both alpha-beta and Monte Carlo do all their reasoning at the level of individual moves. Clearly, humans play games differently: they can reason at a more abstract level, considering a higher-level goal—for example, trapping the opponent’s queen —and using the goal to selectively generate plausible plans. In Chapter 11 we will study this type of planning, and in Section 11.4 we will show how to plan with a hierarchy of abstract to concrete representations. A fourth issue is the ability to incorporate machine learning into the game search process. Early game programs relied on human expertise to hand-craft evaluation functions, opening books, search strategies, and efficiency tricks. We are just beginning to see programs like ALPHAZERO (Silver et al., 2018), which relied on machine learning from self-play rather than � �game-specific human-generated expertise. We cover machine learning in depth starting with Chapter 19 �.Summary We have looked at a variety of games to understand what optimal play means, to understand how to play well in practice, and to get a feel for how an agent should act in any type of adversarial environment. The most important ideas are as follows: A game can be defined by the initial state (how the board is set up), the legal actions in each state, the result of each action, a terminal test (which says when the game is over), and a utility function that applies to terminal states to say who won and what the final score is. In two-player, discrete, deterministic, turn-taking zero-sum games with perfect information, the minimax algorithm can select optimal moves by a depth-first enumeration of the game tree. The alpha–beta search algorithm computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant. Usually, it is not feasible to consider the whole game tree (even with alpha–beta), so we need to cut the search off at some point and apply a heuristic evaluation function that estimates the utility of a state. An alternative called Monte Carlo tree search (MCTS) evaluates states not by applying a heuristic function, but by playing out the game all the way to the end and using the rules of the game to see who won. Since the moves chosen during the playout may not have been optimal moves, the process is repeated multiple times and the evaluation is an average of the results. Many game programs precompute tables of best moves in the opening and endgame so that they can look up a move rather than search. Games of chance can be handled by expectiminimax, an extension to the minimax algorithm that evaluates a chance node by taking the average utility of all its children, weighted by the probability of each child. In games of imperfect information, such as Kriegspiel and poker, optimal play requires reasoning about the current and future belief states of each player. A simple approximation can be obtained by averaging the value of an action over each possible configuration of missing information. Programs have soundly defeated champion human players at chess, checkers, Othello, Go, poker, and many other games. Humans retain the edge in a few games of imperfectinformation, such as bridge and Kriegspiel. In video games such as StarCraft and Dota 2, programs are competitive with human experts, but part of their success may be due to their ability to perform many actions very quickly